

# Introduction to Database Systems

## CSE 444

Lectures 9-10

Transactions: recovery

# Outline

- We are starting to look at DBMS internals
- Today and next time: transactions & recovery
  - Disks 13.2 [Old edition: 11.3]
  - Undo logging 17.2
  - Redo logging 17.3
  - Redo/undo 17.4

# The Mechanics of Disk

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks ( $\leq 10000$ )
- Number of bytes/track( $10^5$ )

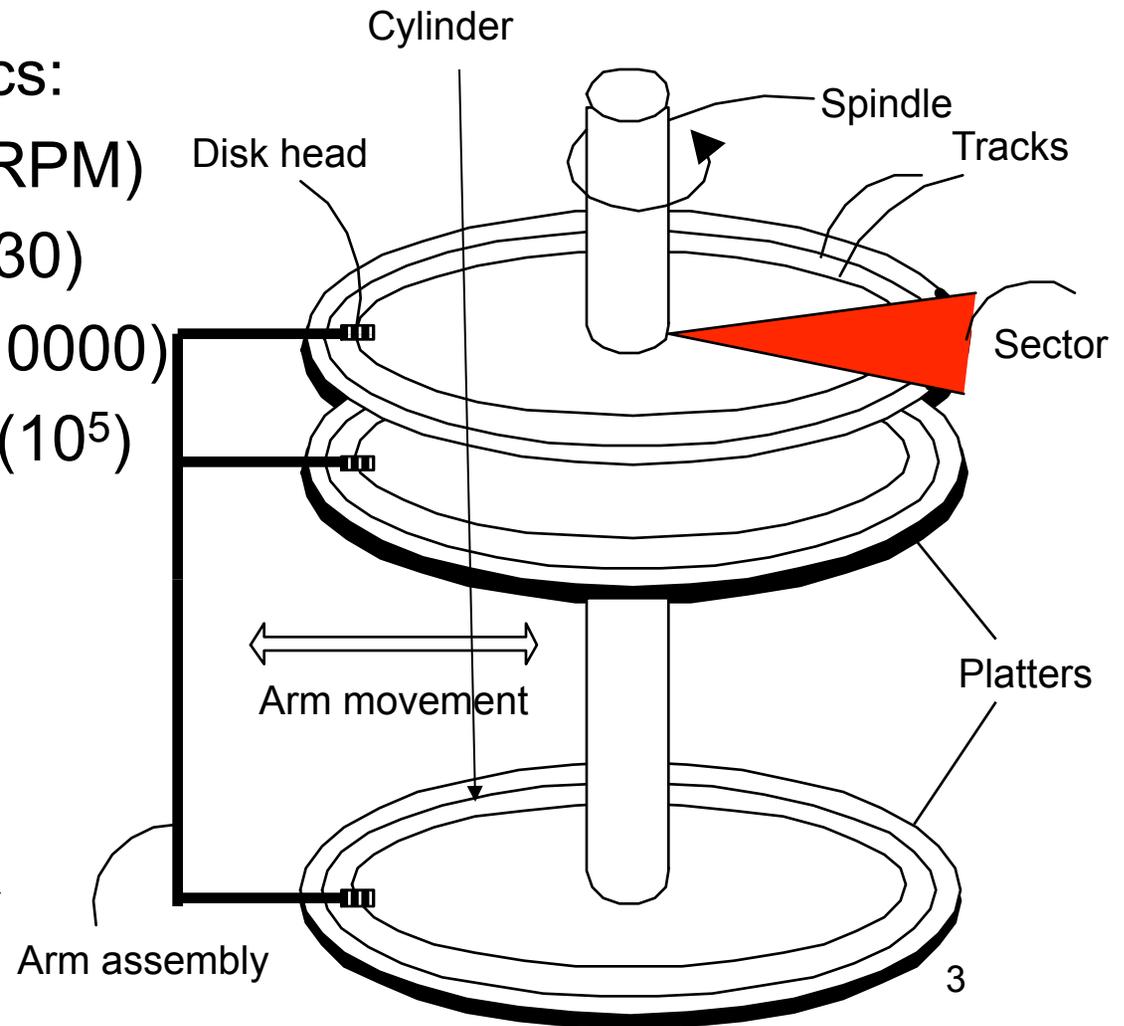
Unit of read or write:

**disk block**

Once in memory:

**page**

Typically: 4k or 8k or 16k



# RAID

Several disks that work in parallel

- Redundancy: use parity to recover from disk failure
- Speed: read from several disks at once

Various configurations (called *levels*):

- RAID 1 = mirror
- RAID 4 =  $n$  disks + 1 parity disk
- RAID 5 =  $n+1$  disks, assign parity blocks round robin
- RAID 6 = “Hamming codes”

Not required for exam, but interesting reading in the book

# Disk Access Characteristics

- **Disk latency** = time between when command is issued and when data is in memory
- Disk latency = seek time + rotational latency
  - Seek time = time for the head to reach cylinder
    - 10ms – 40ms
  - Rotational latency = time for the sector to rotate
    - Rotation time = 10ms
    - Average latency = 10ms/2
- Transfer time = typically 40MB/s
- Disks read/write one block at a time

Large gap between disk I/O and memory → Buffer pool

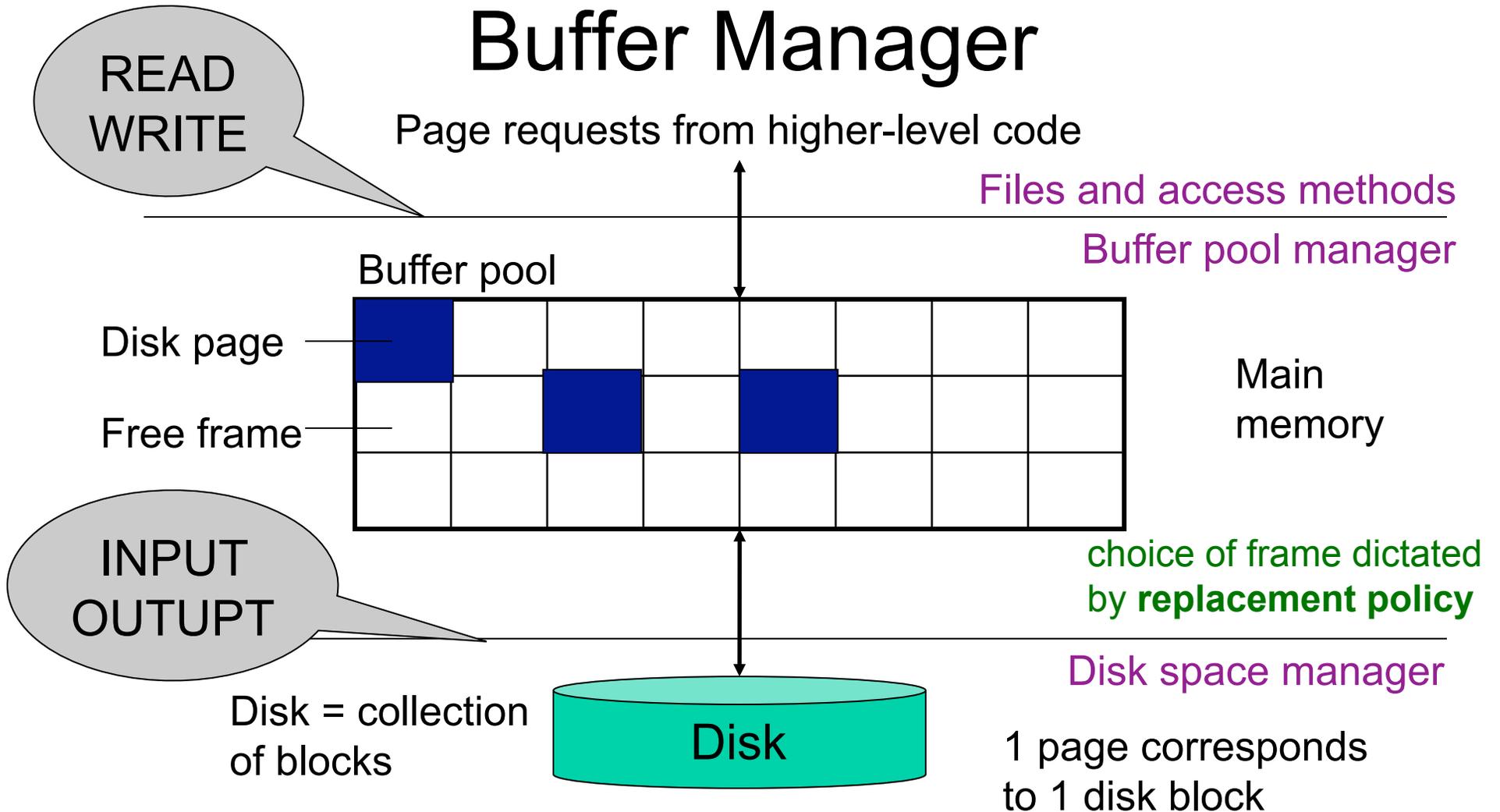
# Design Question

- Consider the following query:

```
SELECT    S1.temp, S2.pressure
FROM      TempSensor S1, PressureSensor S2
WHERE     S1.location = S2.location
AND       S1.time = S2.time
```

- How can the DBMS execute this query given
  - 1 GB of memory
  - 100 GB TempSensor and 10 GB PressureSensor

# Buffer Manager



- Data must be in RAM for DBMS to operate on it!
- Buffer pool = table of <frame#, pageid> pairs

# Buffer Manager

- Enables higher layers of the DBMS to assume that needed data is in main memory
- Needs to decide on page replacement policy
  - LRU = expensive
  - Clock algorithm = cheaper alternative
- Both work well in OS, but not always in DB

# Least Recently Used (LRU)

P5, P2, P8, P4, P1, P9, P6, P3, P7

Read(P6)

P6, P5, P2, P8, P4, P1, P9, P3, ~~P7~~

Read(P10)      Input(P10)

P10, P6, P5, P2, P8, P4, P1, P9, P3

# Buffer Manager

- DBMSs build their own buffer manager and don't rely on the OS. Why?
- **Reason 1: Correctness**
  - DBMS needs fine grained control for transactions
  - Needs to **force** pages to disk for recovery purposes
- **Reason 2: Performance**
  - DBMS may be able to anticipate **access patterns**
  - Hence, may also be able to perform **prefetching**
  - May select better page **replacement policy**

# Transaction Management and the Buffer Manager

Transaction manager operates on buffer pool

- **Recovery**: 'log-file write-ahead', then careful policy about which pages to force to disk
- **Concurrency control**: locks at the page level, multiversion concurrency control

Will discuss details during the next few lectures

# Transaction Management

Two parts:

- Recovery from crashes: ACID
- Concurrency control: ACID

Both operate on the buffer pool

Today, we focus on recovery

# Problem Illustration

Client 1:

```
START TRANSACTION
INSERT INTO SmallProduct(name, price)
  SELECT pname, price
  FROM Product
  WHERE price <= 0.99
```

```
DELETE Product
  WHERE price <=0.99
COMMIT
```

Crash !

## What do we do now?

# Recovery

From which events below can DBMS recover ?

- Wrong data entry
- Disk failure
- Fire / earthquake / etc.
- Systems crashes
  - Software errors
  - Power failures

# Recovery

Type of Crash	Prevention
Wrong data entry	Constraints and Data cleaning
Disk crashes	Redundancy: RAID, backup, replica
Fire or other major disaster	Redundancy: Replica far away
System failures	<b>DATABASE RECOVERY</b>

Most frequent

# System Failures

- Each transaction has *internal state*
- When system crashes, internal state is lost
  - Don't know which parts executed and which didn't
  - Need ability to *undo* and *redo*
- **Remedy: use a log**
  - File that records every single action of each transaction

# Transactions

- Assumption: db composed of **elements**
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)
- Assumption: each transaction reads/writes some elements

# Primitive Operations of Transactions

- **READ(X,t)**
  - copy element X to transaction local variable t
- **WRITE(X,t)**
  - copy transaction local variable t to element X
- **INPUT(X)**
  - read element X to memory buffer
- **OUTPUT(X)**
  - write element X to disk

# Example

```
START TRANSACTION
```

```
READ(A,t);
```

```
t := t*2;
```

```
WRITE(A,t);
```

```
READ(B,t);
```

```
t := t*2;
```

```
WRITE(B,t);
```

```
COMMIT;
```

Atomicity:  
BOTH A and B  
are multiplied by 2

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);



Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)				8	8
READ(A,t)					
t:=t*2					
WRITE(A,t)					
INPUT(B)					
READ(B,t)					
t:=t*2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);

Transaction

Buffer pool

Disk

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)					
t:=t*2					
WRITE(A,t)					
INPUT(B)					
READ(B,t)					
t:=t*2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);



Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)					
INPUT(B)					
READ(B,t)					
t:=t*2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);



Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)					
READ(B,t)					
t:=t*2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);

Transaction

Buffer pool

Disk

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)					
t:=t*2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);

Transaction

Buffer pool

Disk

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);

Transaction

Buffer pool

Disk

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)					
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);

Transaction

Buffer pool

Disk

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)					

READ(A,t); t := t\*2; WRITE(A,t);  
 READ(B,t); t := t\*2; WRITE(B,t);

Transaction

Buffer pool

Disk

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16



Crash occurs after OUTPUT(A), before OUTPUT(B)  
 We lose atomicity

# Buffer Manager Policies

- **STEAL or NO-STEAL**

- Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?

- **FORCE or NO-FORCE**

- Should all updates of a transaction be forced to disk before the transaction commits?

- Easiest for recovery: NO-STEAL/FORCE
- Highest performance: STEAL/NO-FORCE

# Solution: Use a Log

- Log = append-only file containing log records
- Note: multiple transactions run concurrently, log records are **interleaved**
- After a system crash, use log to:
  - **Redo** some transactions that did commit
  - **Undo** other transactions that did not commit
- Three kinds of logs: undo, redo, undo/redo

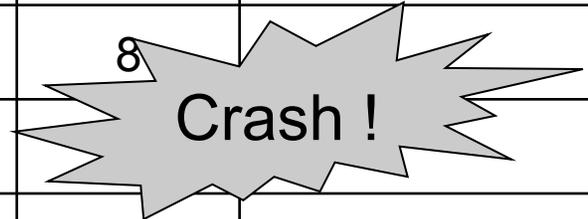
# Undo Logging

## Log records

- **<START T>**
  - Transaction T has begun
- **<COMMIT T>**
  - T has committed
- **<ABORT T>**
  - T has aborted
- **<T,X,v>** -- Update record
  - T has updated element X, and its old value was v

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>



WHAT DO WE DO ?

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

WHAT DO WE DO ?



# After Crash

- In the first example:
  - We UNDO both changes:  $A=8$ ,  $B=8$
  - The transaction is atomic, since none of its actions has been executed
- In the second example
  - We don't undo anything
  - The transaction is atomic, since both its actions have been executed

# Undo-Logging Rules

U1: If T modifies X, then  $\langle T, X, v \rangle$  must be written to disk before OUTPUT(X)

U2: If T commits, then OUTPUT(X) must be written to disk before  $\langle \text{COMMIT } T \rangle$

- Hence: OUTPUTs are done early, before the transaction commits

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

# Recovery with Undo Log

After system's crash, run recovery manager

- Idea 1. Decide for each transaction T whether it is completed or not
  - <START T>....<COMMIT T>.... = yes
  - <START T>....<ABORT T>..... = yes
  - <START T>..... = no
- Idea 2. Undo all modifications by incomplete transactions

# Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - <COMMIT T>: mark T as completed
  - <ABORT T>: mark T as completed
  - <T,X,v>: if T is not completed  
          then write X=v to disk  
          else ignore
  - <START T>: ignore

# Recovery with Undo Log

```
...  
...  
<T6,X6,v6>  
...  
...  
<START T5>  
<START T4>  
<T1,X1,v1>  
<T5,X5,v5>  
<T4,X4,v4>  
<COMMIT T5>  
<T3,X3,v3>  
<T2,X2,v2>
```

crash

Question 1 in class:  
Which updates are  
undone ?

Question 2 in class:  
What happens if there  
is a second crash,  
during recovery ?

Question 3 in class:  
How far back  
do we need to  
read in the log ?

# Recovery with Undo Log

- Note: all undo commands are *idempotent*
  - If we perform them a second time, no harm done
  - E.g. if there is a system crash during recovery, simply restart recovery from scratch

# Recovery with Undo Log

When do we stop reading the log ?

- We cannot stop until we reach the beginning of the log file
- This is impractical

Instead: use checkpointing

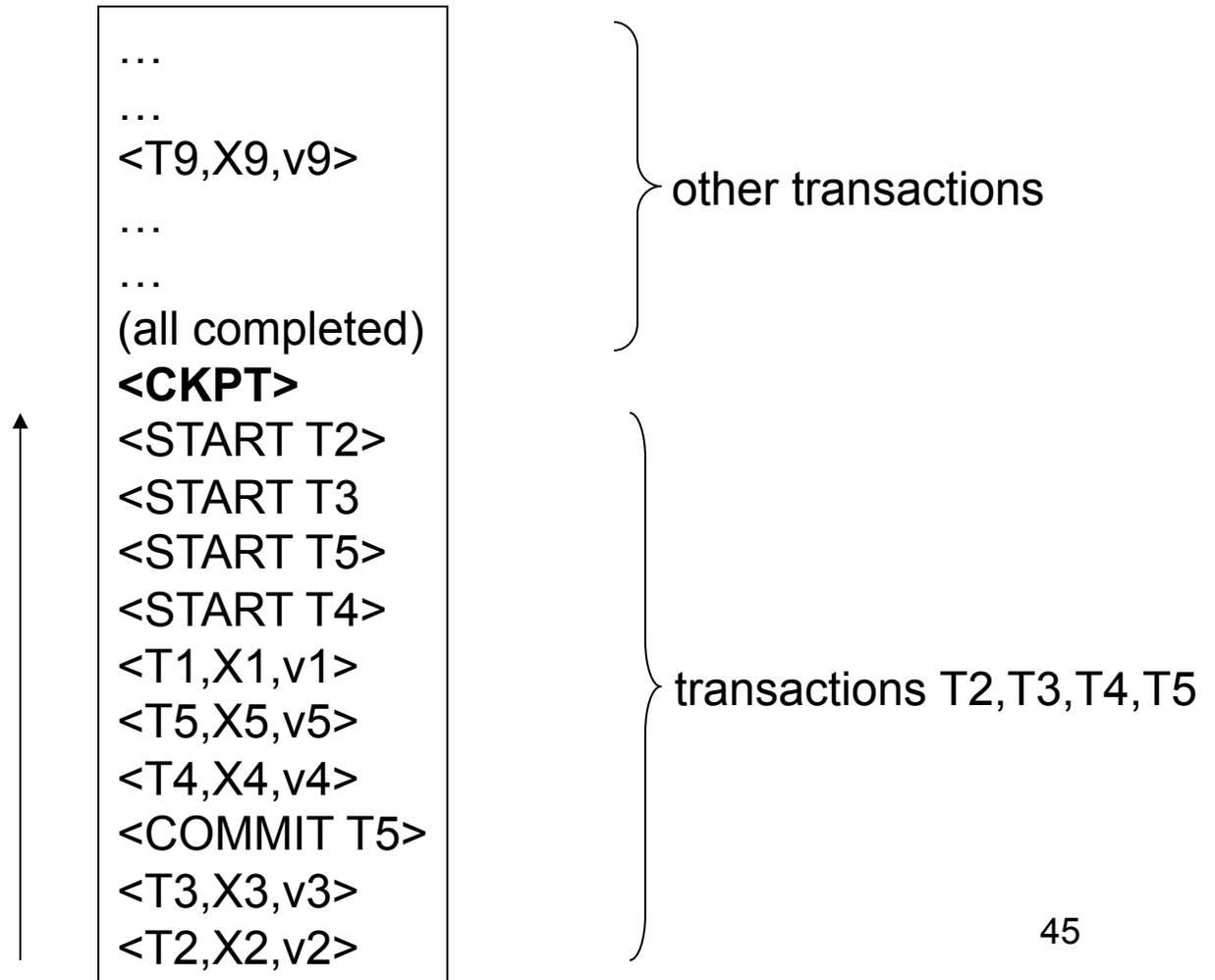
# Checkpointing

Checkpoint the database periodically

- Stop accepting new transactions
- Wait until all current transactions complete
- Flush log to disk
- Write a <CKPT> log record, flush
- Resume transactions

# Undo Recovery with Checkpointing

During recovery,  
Can stop at first  
<CKPT>



# Nonquiescent Checkpointing

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- Idea: nonquiescent checkpointing

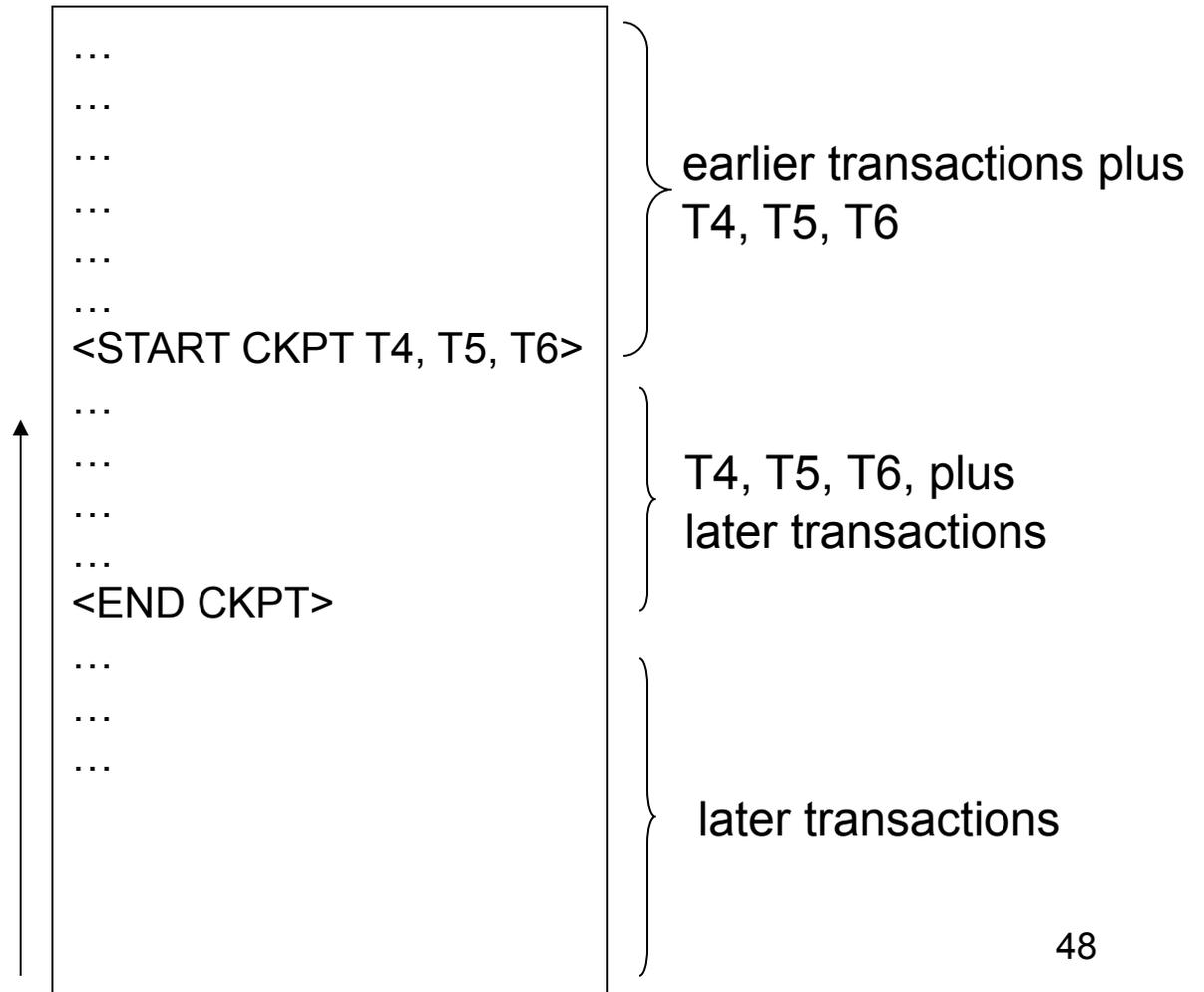
Quiescent = being quiet, still, or at rest; inactive  
Non-quiescent = allowing transactions to be active

# Nonquiescent Checkpointing

- Write a  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  where  $T_1, \dots, T_k$  are all active transactions. Flush log to disk
- Continue normal operation
- When all of  $T_1, \dots, T_k$  have completed, write  $\langle \text{END CKPT} \rangle$ . Flush log to disk

# Undo Recovery with Nonquiescent Checkpointing

During recovery,  
Can stop at first  
<CKPT>



Q: why do we need  
<END CKPT> ?

# Implementing ROLLBACK

- Recall: a transaction can end in COMMIT or ROLLBACK
- Idea: use the undo-log to implement ROLLBACK
- How ?
  - LSN = Log Sequence Number
  - Log entries for the same transaction are linked, using the LSN's
  - Read log in reverse, using LSN pointers

# Redo Logging

## Log records

- $\langle \text{START } T \rangle$  = transaction  $T$  has begun
- $\langle \text{COMMIT } T \rangle$  =  $T$  has committed
- $\langle \text{ABORT } T \rangle$  =  $T$  has aborted
- $\langle T, X, v \rangle$  =  $T$  has updated element  $X$ , and its new value is  $v$

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

# Redo-Logging Rules

R1: If T modifies X, then both  $\langle T, X, v \rangle$  and  $\langle \text{COMMIT } T \rangle$  must be written to disk before  $\text{OUTPUT}(X)$

- Hence: OUTPUTs are done late

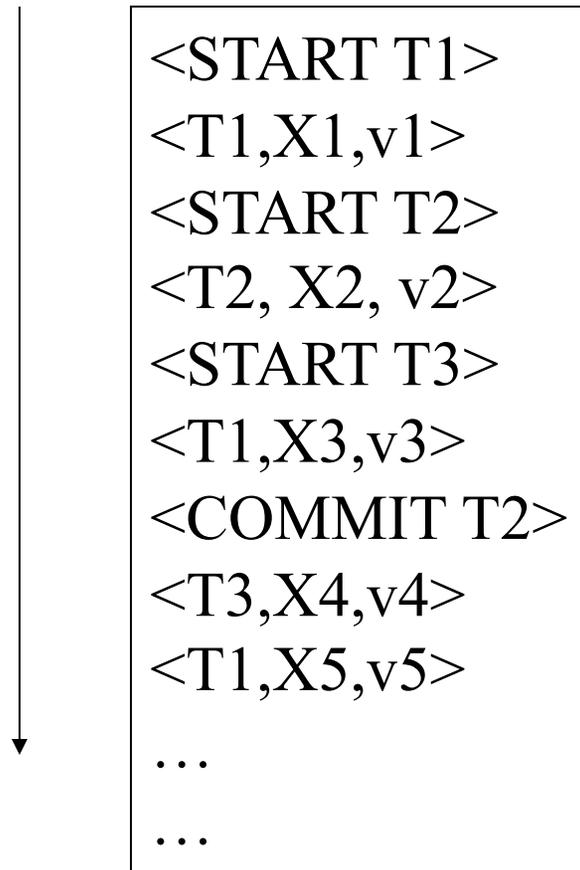
Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

# Recovery with Redo Log

After system's crash, run recovery manager

- Step 1. Decide for each transaction T whether it is completed or not
  - <START T>....<COMMIT T>.... = yes
  - <START T>....<ABORT T>..... = yes
  - <START T>..... = no
- Step 2. Read log from the beginning, redo all updates of committed transactions

# Recovery with Redo Log



# Nonquiescent Checkpointing

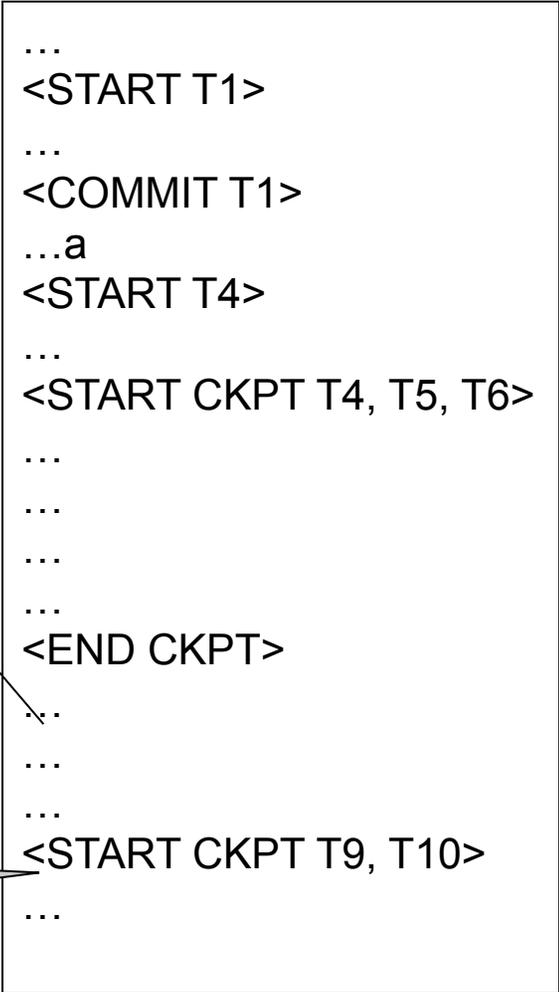
- Write a  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  where  $T_1, \dots, T_k$  are all active transactions
- Flush to disk all blocks of committed transactions (*dirty blocks*), while continuing normal operation
- When all blocks have been written, write  $\langle \text{END CKPT} \rangle$

# Redo Recovery with Nonquiescent Checkpointing

Step 1: look for  
The last  
<END CKPT>

All OUTPUTs  
of T1 are  
known to be on disk

Cannot  
use



Step 2: redo  
from the  
earliest  
start of  
T4, T5, T6  
ignoring  
transactions  
committed  
earlier

# Comparison Undo/Redo

- **Undo logging:**
  - OUTPUT must be done early
  - If <COMMIT T> is seen, T definitely has written all its data to disk (hence, don't need to redo) – inefficient

Steal/Force
- **Redo logging**
  - OUTPUT must be done late
  - If <COMMIT T> is not seen, T definitely has not written any of its data to disk (hence there is not dirty data on disk, no need to undo) – inflexible

No-Steal/No-Force
- Would like more flexibility on when to OUTPUT:  
**undo/redo logging** (next)

Steal/No-Force

# Undo/Redo Logging

Log records, only one change

- $\langle T, X, u, v \rangle =$  T has updated element X, its old value was u, and its new value is v

# Undo/Redo-Logging Rule

UR1: If  $T$  modifies  $X$ , then  $\langle T, X, u, v \rangle$  must be written to disk before  $\text{OUTPUT}(X)$

Note: we are free to  $\text{OUTPUT}$  early or late relative to  $\langle \text{COMMIT } T \rangle$

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8,16>
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

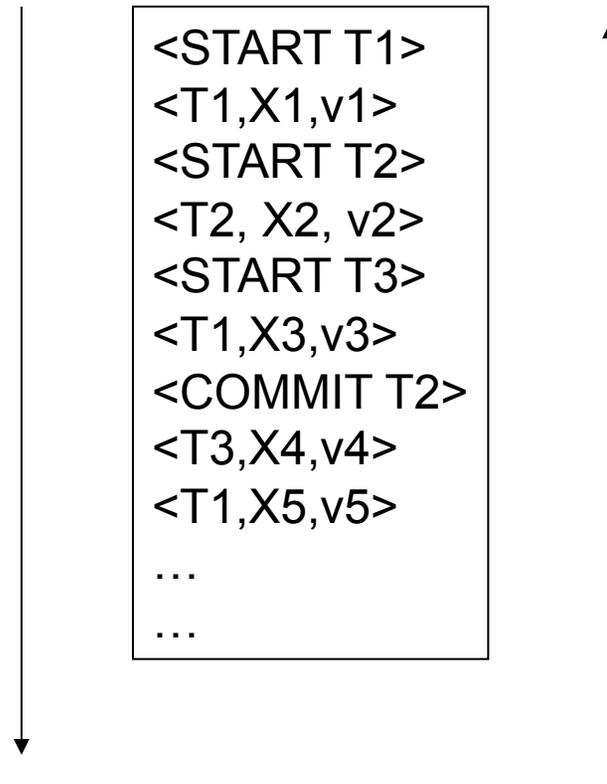
Can OUTPUT whenever we want: before/after COMMIT

# Recovery with Undo/Redo Log

After system's crash, run recovery manager

- Redo all committed transaction, top-down
- Undo all uncommitted transactions, bottom-up

# Recovery with Undo/Redo Log



# Granularity of the Log

- Physical logging: element = physical page
- Logical logging: element = data record
  
- What are the pros and cons ?

# Granularity of the Log

- Modern DBMS:
- Physical logging for the REDO part
  - Efficiency
- Logical logging for the UNDO part
  - For ROLLBACKs